Out-Of-Core Streamline Visualization on Large Unstructured Meshes

1997

Shyh-Kuang Ueng†
K. Sikorski
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Kwan-Liu Ma†
ICASE
Mail Stop 403
NASA Langley Research Center
Hampton, Virginia 23681

Abstract

It's advantageous for computational scientists to have the capability to perform interactive visualization on their desktop workstations. For data on large unstructured meshes, this capability is not generally available. In particular, *particle tracing* on unstructured grids can result in a high percentage of non-contiguous memory accesses and therefore may perform very poorly with virtual memory paging schemes. The alternative of visualizing a lower resolution of the data degrades the original high-resolution calculations.

This paper presents an out-of-core approach for interactive streamline construction on large unstructured tetrahedral meshes containing millions of elements. The out-of-core algorithm uses an octree to partition and restructure the raw data into subsets stored into disk files for fast data retrieval. A memory management policy tailored to the streamline calculations is used such that during the streamline construction only a very small amount of data are brought into the main memory on demand. By carefully scheduling computation and data fetching, the overhead of reading data from the disk is significantly reduced and good memory performance results. This out-of-core algorithm makes possible interactive streamline visualization of large unstructured-grid data sets on a *single* mid-range workstation with relatively low main-memory capacity: 5-20 megabytes. Our test results also show that this approach is much more efficient than relying on virtual memory and operating system's paging algorithms.

†This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

1 Introduction

Most visualization software tools have been designed for data that can fit into the main memory of a single workstation. For many scientific applications, data at the desirable accuracy overwhelm the memory capacity of the scientist's desk-top workstation. This is particularly true for data obtained from three dimensional aerodynamics calculations, where very fine unstructured tetrahedral meshes are needed to model arbitrarily complex configurations such as an airplane. Although adaptive meshing techniques can be applied to reduce the resolution of the meshes, the resulting meshes may contain tens of millions of tetrahedral cells.

Rapidly increasing CPU performance and memory capacity are beginning to allow scientist to study data at such resolutions. Many scientists now have access to workstations with 500 megabytes to one gigabyte of memory which are capable of visualizing millions of tetrahedral cells. But the same capability also allows scientists to model problems at even greater resolution. Moreover, not every scientist has constant access to such high-end workstations.

To solve this problem, previous research has mainly focused on the use of parallel and distributed computers, and multiresolution data representations. For example, pV3 [3] (parallel Visual3) breaks up the problem domain in space and places each partition on an individual workstation; streamlines are then calculated in a distributed, interactive manner. In particular, pV3 can couple visualization calculations with the simulation. This approach is very attractive to scientists in an open, distributed computing environment and has also been shown to work well on a distributed-memory parallel computer like the IBM SP2 [4].

Another popular approach is to make use of a supercomputer like a CRAY for visualization calculations and a high-end graphics workstation for displaying the streamlines. For streamline visualization, this approach is preferable to the distributed approach since streamline calculations do not parallelize well. Finally, multiresolution data representations allow the user to explore the data at a lower resolution according to the computer's performance, but they are still memory-limited at the highest resolutions.

More recently, visualization software companies [1] as well as corporate research laboratories [11] have begun to look into this problem and attempted to provide viable solutions for their software products. While their solutions might be for more advanced graphics workstations and more general visualization purpose, ours, an *out-of-core* approach, intends to enable interactive *streamline* visualization of large unstructured grid data on mid-range workstations or even PC-class machines with only a moderate amount of main memory.

1.1 Why Out-Of-Core?

Out-of-core processing is not new and in fact has long been used to cope with large data. Many computational problems in engineering and science involve the solution of an extremely large linear system that does not fit into a computer's main memory. Using an out-of-core method is the only solution in the absence of large memory space and parallel computers. Another example is from database applications; a large database can only be constructed with an out-of-core approach.

Will an operating system be smart enough to handle memory contention caused by using brute-force algorithms for data visualization, solving linear systems, or database construction? The answer is no. Modern operating systems are good at managing multiple jobs and providing time sharing via paging and swapping. But they cannot make more memory appear out of nowhere. In particular, when data access is random and irregular, typical in unstructured data visualization, poor locality of referencing leads to thrashing in the virtual memory.

For example, unstructured-grid data generally store coordinates and solution values for each node (a grid point), and node indices for both triangular faces and tetrahedral cells. As shown in Figure 1, these node, face and cell data are not stored contiguously in disk space according to their spatial relationship. During visualization calculations, accessing two neighboring cells may invoke references farther apart in disk space. Consequently, constant paging is forced to fetch disk resident data and this memory overload eventually becomes I/O overload.

While moderate paging is common, desperation swapping is often intolerable. It has been evident that many commercial and free visualization software packages fail to handle large data sets on an average workstation. This research has been motivated by our local scientists' need of an interactive visualization mechanism to study their data at the desirable resolution, and *particle tracing* is one of the most important capabilities requested.

1.2 An Out-Of-Core Streamline Visualization Algorithm

Streamlines are the paths of massless particles released in steady flow fields [15]. Plotting streamlines is a fundamental technique for visualizing vector field data sets generated from scientific computations [7, 9, 14]. Streamlines can be extended to construct other types of objects, like streamtubes and streamribbons [2, 5, 14]. A streamline is usually constructed by using stepwise numerical integration. The integration involves the following steps:

1. Selecting of an initial point.

2. Locating the cell containing the point.

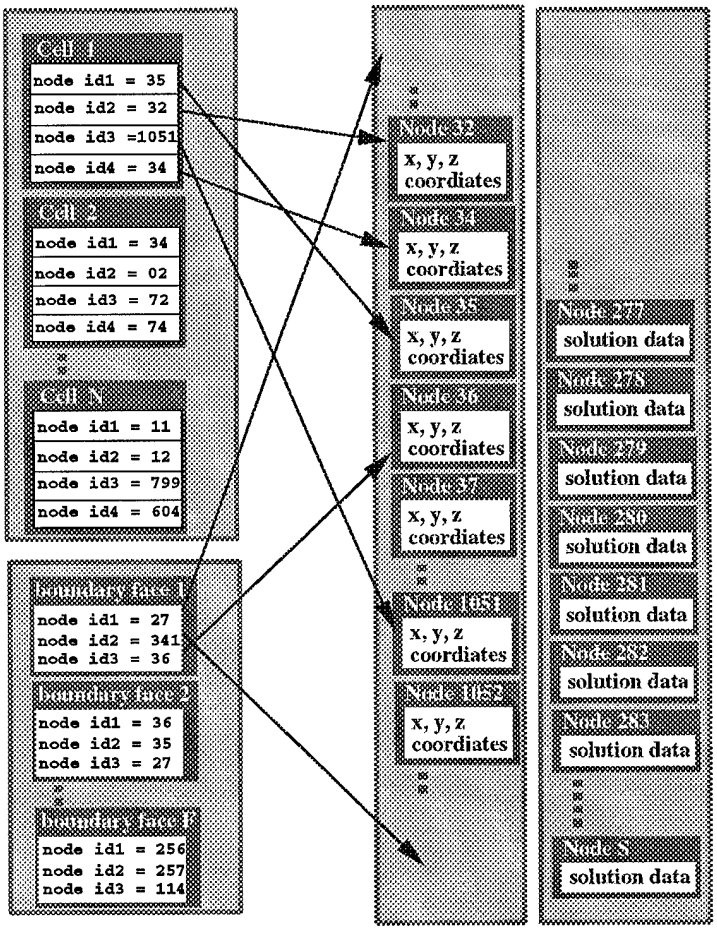

Figure 1: A typical data structure for unstructured meshes. Normally, node, face, and cell data are stored as separate chunks. Accessing two cells next to each other in the spatial domain may invoke references to the corresponding data items scattering across the disk space.

3. Interpolating the vector field and calculating a new point by using a numerical integration method.

4. If the termination condition is not met, go to step 2.

Our out-of-core algorithm has been designed based on the following observations:

- Streamline calculations are incremental and local. Each integration step only needs a very small amount of data, one or two tetrahedral cells.
- Calculating multiple streamlines concurrently is cheaper than calculating one streamline after the other. This maximizes locality of reference which increases memory performance dramatically.
- Data packing is essential to reduce the number of disk reads. Data should be packed in such a way that fetching cells in a small neighborhood can be done with one disk read.
- It is much more efficient to read small chunks of data from disk. Moving a larger chunk of data from disk would likely disrupt the interactivity when a streamline is ready to enter a neighboring chunk.

The resulting algorithm contains two steps: preprocessing and interactive streamline construction. The preprocessing step determines connectivity, calculates additional quantities such as interpolation functions and coordinates transformation functions, restructures the raw data, and stores all the information into a more compact *octree* representation on disk. This step needs to be done only once. The second step requires a graphical user interface to facilitate picking of seed points where tracing of streamlines begins. The interactive streamline construction step does not rely on the operating system to fetch the required data. Instead, a memory management policy is designed to efficiently utilize a minimum memory space and fetch data from disk. Streamlines are integrated from octants to octants based on the principles of *preemption* and *time-sharing*. In this way, streamlines can be constructed interactively by using only a few megabytes of memory space on a mid-range workstation like a Sun SPARC-20.

The rest of the paper is organized as follows. Section 2 illustrates the data preprocessing step. The streamline construction algorithm is described in Section 3 and the memory management policy is explained in Section 4. Tests are performed to compare virtual memory against our algorithm; to study the performance of the memory management policy, local disk access and non-local disk access; and to measure average cost and overhead. The test results are presented and discussed in Section 5, followed by some concluding remarks and future research directions.

2 Data Preprocessing

Efficient visualization operations on unstructured-grid data can only be obtained with preprocessing because of the irregularity of the mesh topology. To perform streamline visualization, the two most important operations are:

1. identifying the tetrahedra cell containing the user specified seed point.

2. computing velocity at locations other than the node points.

Fast cell searching methods like the one presented in [9] need additional data such as cell connectivity information and coordinate transformation functions. These data, which are also needed by the integration step, could be computed on the fly during streamline construction, but the computational cost would then be too high for interactive visualization.

As flow solutions are only defined at node locations, interpolation must be used to compute flow variable values at other locations. To attain maximum efficiency at run time, an interpolation function for each cell is also precomputed and stored with the data. For tetrahedral cells, we use the *linear basis function interpolation* [6, 14]. In summary, our preprocessing step first determines cell connectivity; then computes the coordinate transformation and interpolation functions for each cell; and finally partitions and reorganizes the raw data with the computed data using an octree structure to facilitate fast data retrieval. To achieve interactive visualization, we cannot avoid precomputing and storing some of these data. The additional storage space required actually makes the out-of-core approach even more attractive. The issues and techniques for calculating transformation and interpolation functions as well as connectivity information can be found in previous research [13, 14].

2.1 Data Partitioning

There are two approaches for partitioning unstructured data sets. The first approach is to divide cells into totally disjoint groups. Since the data sets are unstructured, the geometric shapes of the resulting groups are generally irregular. The advantage of this approach is that no data redundancy is introduced. However, one of the disadvantages is that the spatial relationship between groups is difficult to determine. Specifically, it becomes difficult to verify whether two groups are adjacent, and to identify the group where a specified point is located.

The second approach is to partition the data set by superimposing a regular framework on it. A subset is formed by grouping the cells which intersect or are contained within a region of the framework. The framework could be a regular 3-D mesh, a *k-way tree* or an *octree* [10]. Since the data sets are unstructured, a cell may intersect with several regions of a regular framework and thus data redundancy is inevitable with this approach.

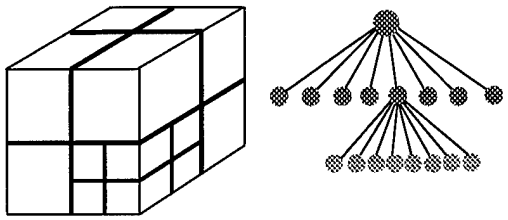

Figure 2: Octree Data Partitioning.

A major advantage of the second approach is that the spatial information of subsets can be easily obtained. For example, if an *octree* is employed as the framework of data partition, the octant containing the seed point can be identified by searching the octree from the root to the leaves within $O(\log N)$ steps, where N is the number of the octants. The neighbors of an octant can also be found by applying this technique and one of these neighboring octants contains the next point on the current streamline.

In our out-of-core setting, octrees are used as the framework for the data partitioning since unstructured grids are highly *adaptive* in both shape and resolution. Octrees have been widely employed by many computer graphics and visualization applications. It allows us to refine the data partitioning in the regions where the grids are dense such that subsets are relatively equal in data size (i.e. in terms of the number of tetrahedral cells). In Figure 2, a simple example of octree is shown. Note if the framework is a regular 3-D mesh, the above searches may be completed in constant time. But a very high resolution regular mesh must be used to accommodate the original mesh's irregularity.

Based on an octree structure, the data partitioning is carried out in a top-down manner. First, the whole data set is considered as one octant. Then this octant is decomposed into eight child octants by using three cutting planes perpendicular to the x, y, and z axes. If the number of tetrahedral cells in a child octant exceeds a pre-defined limit, the *maximum octant size*, this child octant is partitioned further. The above procedure is performed recursively until all octants contain fewer cells than the maximum octant size. The cells of an octant are stored in a file in our current implementation. This enables very straightforward access to an octant, though a large number of files may be created when the *maximum octant size* is small. An alternative way is to store all octants in a single file. This method must employ an indexing algorithm if the sizes of octants are different and the number of octants is large.

Each octant stores the bounding box of the octant, the number of cells in the octant, the center of the octant, and the ID of the file used for storing cells in that octant as shown in Figure 3. The center of the octant is where the three cutting planes intersect. The position of this point is set to be the arithmetic average of the centers of all cells in the octant, where

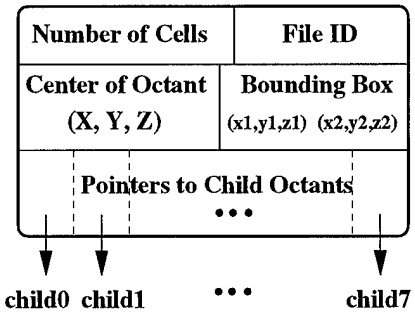

Figure 3: Data Structure of an Octree Node

the *center* of a cell is defined as the arithmetic average of its vertices. This choice keeps the size of the eight child octants at each level of the tree about the same.

The octree created represents the structure of the data partitions and is stored in a file after the partitioning is completed. This file is read in first at the beginning of a streamline visualization session. A typical octree requires under one megabyte of storage space. The structure of the octree nodes allows an efficient, systematic way of retrieving the needed data for calculating the streamlines.

2.2 Out-Of-Core Data Preprocessing

For the sizes of data we consider, the data preprocessing step must also be performed in an out-of-core manner. This is done by allocating eight buffers in memory and opening eight disk files to store cells read from the input data file. At the top level, these eight buffers and disk files correspond to the root's eight child octants and their bounding regions. Then cells are read into memory incrementally and a cell is assigned to a buffer if it intersects the corresponding bounding region. As mentioned previously, a cell may be assigned to more than one buffer. Whenever a buffer is full, the cells in the buffers are dumped to the corresponding disk file. After all cells are processed, each octant size is examined. If an octant has more cells than the *maximum octant size*, another round of partitioning proceeds. Eight more buffers and disk files are created.

After the octree is completely built, the next step is to find cell connectivities and calculate the coefficients of the coordinate transformation as well as interpolation functions. One octant file is processed at a time. Note that the *maximum octant size* determines the number of octants generated. A larger octant size implies less data redundancy and thus less disk space used. But the problem with keeping large octants in the main memory is that it is then harder to achieve consistent performance. Remember that for streamline visualization moving many smaller data chunks is generally less expensive than moving a

few larger pieces since normally only a small portion of each data chunk is accessed by the streamline calculation. Moreover, a higher hit rate would be achieved with many smaller octants in core. On the other hand, if the maximum octant size is relatively small, a larger number of octants are generated. The preprocessing step would become more expensive. The data redundancy becomes higher, and more disk space is required to store the data. However, more octants can be resident in the main memory during streamline constructions to attain more consistent performance. Test results will be provided in section 5 to show how selections of the *maximum octant size* influence the performance of the out-of-core method.

3 Streamline Construction

Two operations are repeatedly performed during integrating streamlines. The first one is to compute new positions of streamlines and the second one is to move the required data from disk into main memory if it is not already there. Compared with CPU speed and memory access time, disk I/O is relatively slow. In order to narrow the gap, the computation and the data-fetching have to be carefully scheduled. Furthermore, the memory space is a limited resource. It is important to fully utilize the memory space to store more information for calculation such that computation can be carried out with minimum interruption.

In order to achieve these goals, the out-of-core streamline construction algorithm is based on two fundamental operating system concepts: *preemption* and *time-sharing* [12]. Based on the availability of the data, a streamline under construction may be in any of the following three states: *waiting, ready*, or *tracing*. When the needed octant is in the main memory, the streamline is in the ready state, and it can enter the tracing state; that is, its next positions can be calculated. Otherwise, the streamline is in the waiting state, waiting for the needed octant to be brought in from the disk. When the memory space occupied by an octant is no longer involved in computing new streamline positions, it can be released and reused.

In short, the out-of-core program following the preprocessing stage consists of the following steps:

- Initialization:
  - Read the octree created in the data partitioning step from the disk.
  - Allocate memory space for holding octants.
  - Create data structures needed in the streamline construction.
- Construction of the streamlines:
  1. Get the initial positions selected by the user.
  2. Identify the octants where the streamlines enter.

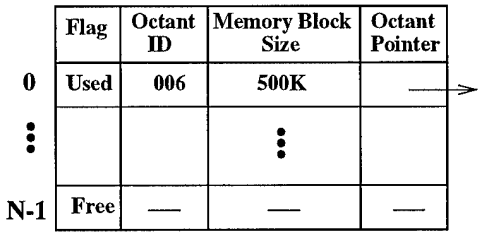

Figure 4: An Octant Table

3. Fetch octants into the main memory.
4. Integrate all streamlines with their octants in the main memory until all of them leave the octants.
5. Go back to 2, if a termination condition for any of the streamlines is not met.

3.1 Initialization

The initialization step first reads the octree from the disk and creates the following data structures:

- an octant table to keep track of the octants in the main memory,
- three queues for scheduling computations.

The *octant table* is used to store information about octants which are resident in main memory. One octant is associated with each entry in the octant table. Each entry contains four fields. Figure 4 shows a table of N entries. The first field is a flag indicating whether this entry is allocated to an octant or not. The second field contains the ID of the octant. The third field stores the size of the memory space allocated to the octant. The last one is a pointer to the starting position of the octant.

Three queues are created to keep data about the streamlines under construction. These queues are named the *waiting queue*, the *ready queue*, and the *finished queue*. A streamline is kept in the waiting queue if it enters an octant which is not resident in the main memory. Otherwise, it is in the ready queue. Once the streamline is completely integrated, it is stored in the finished queue. These three queues and the octant table are employed to schedule streamline construction and octant-fetching such that more streamlines can be processed at the same time by using less memory space.

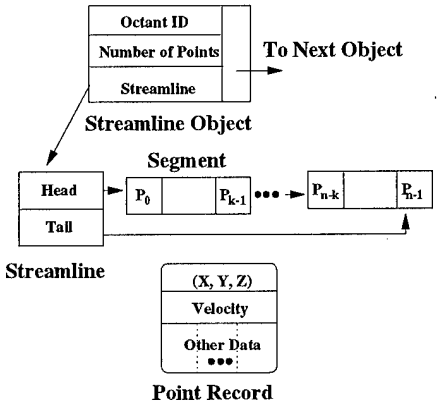

Figure 5: Data Structures of a Streamline

3.2 Construction

Given an initial seed point, a *streamline object* is created. The streamline object stores a list of streamline positions, the number of points in the streamline and the ID of the octant containing the most recently integrated position of the streamline. For each streamline point, the coordinates, the velocity magnitude, the *angular rotation rate of the flow*, and the *local flow expansion rate* [14] are recorded. The data structure of a streamline object is depicted in Figure 5. Initially, the ID's of the octants which contain the initial positions are identified and entered into the streamline objects, and all streamline objects are kept in the waiting queue.

In the next step of the streamline construction, the streamline objects in the waiting queue are examined one by one. As long as there is still space in the pre-allocated main memory, the octant identified by a streamline object is read into the octant table. Once the octant of a streamline object has been read, the streamline object is moved from the waiting queue to the ready queue.

Subsequently, the streamline objects in the ready queue are processed one by one. The fourth order Runge-Kutta method [8] is used to calculate new streamline points. At the same time, the angular rotational rate and the local flow expansion rate are computed if streamribbons and streamtubes are to be constructed [13]. When a streamline leaves an octant, the octant containing the new position of the streamline is identified. Then the octant table is searched to check whether this octant is already in the main memory or not. If it is, the data cell containing the current streamline position is found, and the streamline construction continues. If it is not, the streamline object is moved to the end of the waiting queue, and another streamline object is selected from the ready queue for processing. If the

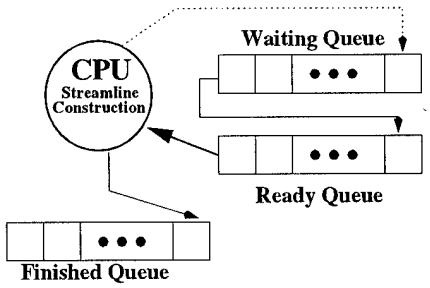

Figure 6: Streamline Object Scheduling

streamline reaches a physical domain boundary or its current time step exceeds a pre-defined limit, the streamline object is deleted from the ready queue and stored in the finished queue.

Once the ready queue is empty, the octants in the main memory are no longer involved in the streamline construction. The memory space occupied by them is released to a free space pool. Their octant table entries are marked as free. The waiting queue is searched, and a new set of octants is fetched into the main memory. Then another round of streamline construction begins. The streamline integration is completed when all streamline objects are in the finished queue. An example illustrating the migration of streamline objects during the streamline construction is depicted in Figure 6.

4 Memory Management

The octants produced in the preprocessing stage may have different sizes. It is therefore unwise to use a fixed size for memory blocks, each of which holds an octant. For efficient utilization of the memory space, a memory management policy is designed to support the out-of-core streamline visualization program. First, the size of the memory space dedicated to the out-of-core program is selected by the user. Presently, this size is measured in number of cells, and it should be greater than the maximum octant size. This memory space is decomposed into memory blocks of different sizes. The size of the memory blocks created is determined by two other parameters: the maximum octant size and a parameter called the *block size level*. These two values can be either controlled by the user or set automatically based on information obtained from the preprocessing step. The block size level represents the number of different block sizes. For example, if its value is one, then all blocks are of the same size, which is equal to the maximum octant size. If it is set to k, the sizes of blocks are $\frac{s}{n}$ where s is the maximum octant size and $n = 1, ..., k$.

The blocks are created in a descending order of sizes; that is, the largest block is generated first, then a block of the second largest size is created, and so on. During the creation process,

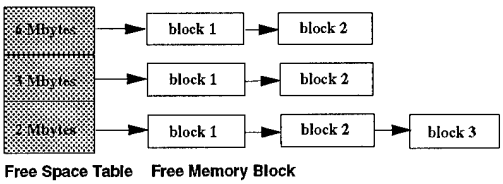

Figure 7: Free Memory Space Pool

if the remaining memory space is too small for creating a block of a particular size, this size is skipped, and a block of the next smaller size is to be created. However, if the remaining memory space is smaller than the smallest block size, then the process stops. If the smallest block is created, then we re-run the process if the remaining memory space is large enough for creating any blocks. All memory blocks created are then put into a *free space pool*. In this pool, a table is created for book-keeping. The number of entries in this table equals to the block size level. In each entry, a list of blocks of the same size is maintained. An example of a free space pool is shown in Figure 7, in which the block size level is three.

Before an octant is fetched into the main memory, the size of the octant is retrieved from the octree. The free space pool is searched to find a memory block that is large enough to hold the octant. This searching starts at the list of the smallest blocks such that a *best-fit* block may be found. Once a block is assigned to an octant, it is removed from the free space pool. When this octant is no longer involved in computations, the memory block is released to the free space pool.

The block size level is an important parameter determining memory utilization efficiency. It cannot be too small or too large; while the former results in a few large blocks which are space inefficient to store smaller octants and would cause excessive octant fetching, the latter results in many smaller blocks which might be too small and therefore never used. Some tests have been run to study the effects of this parameter upon the out-of-core program. The results will be presented in the next section.

5 Test Results

We tested the out-of-core visualization algorithm on an IBM RS6000 workstation with 128 megabytes of main memory as well as a Sun SPARC-20 workstation with 64 megabytes of main memory. Note that our algorithms only need about 5-20 megabytes out of the 64/128 megabytes to achieve interactive visualization. The IBM workstation with larger memory space allows us to compare the performance of the out-of-core method with programs relying on virtual memory management. In addition, three sets of tests are conducted on the Sun workstation. The first set of tests are used to reveal how the *maximum octant size*, the

memory space size and the *block size level* affect the overall performance of the out-of-core program. In the second set of tests, the overhead produced by fetching data and scheduling computations is recorded and analyzed. The third set studies the effect caused by storing data in a non-local disk.

In all tests, wall clock time is used to measure the cost. All tests were run in batch mode and rendering and display time is not included. Currently, rendering is done in software but the fast streamline construction rate and incremental software rendering make interactive viewing of streamline formation possible.

5.1 The Out-Of-Core Method versus Virtual Memory

In order to reveal the strength of the out-of-core method, two streamline construction methods that rely on virtual memory are implemented for testing. All three programs use the same numerical method to integrate streamlines. The two virtual-memory-based methods attempt to store as much data as possible in the main memory. In the first program, a cell record contains four vertex indices and four neighboring cell indices. The size of a cell record is 32 bytes. The four neighboring cell indices of each cell record are calculated in a preprocessing stage, though the coefficients of the coordinate transformation function are computed on the fly during streamline construction. In the second program, a cell record stores four vertex indices, four neighboring cell indices and a 3×4 matrix in which the coordinate transformation function coefficients are stored. That is, eight integers and 12 floats are kept in a cell record so the size of a cell record is 80 bytes. The neighboring cell indices and the coordinate transformation function coefficients are pre-computed in the preprocessing stage.

In the out-of-core program, each cell record holds the same information as the second program. The maximum octant size is set to 20,000 cells, and a memory space that is equal to six times of the maximum octant size is dedicated to the program. The block size level is set to three.

The data sets of the tests are artificially created by dividing a cube into one, two, three, four, and five million tetrahedral cells. For all these sets, the memory requirement for storing streamlines, vertices, and cell records is larger than the user space of the main memory. For example, four million tetrahedral cells would require at least 128 megabytes while the dedicated user space is under ten megabytes.

To generate the artificial data sets, the three components of the vector field are determined by the following formula:

$$\begin{aligned} u(x,y,z) &= -0.5x - 6.0y, \\ v(x,y,z) &= 6.0x - 0.5y, \\ w(x,y,z) &= -2.0z + 20.5. \end{aligned}$$

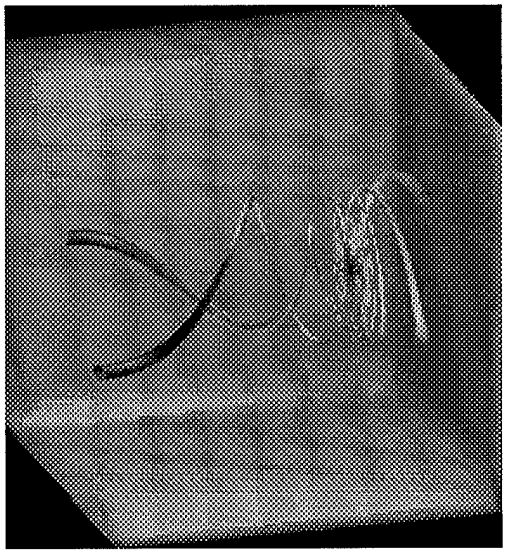

Figure 8: Streamline visualization of the artificial data set.

Table 1: VM-Based Method 1

data size	initiate	construct	total
1M	15.40	917.08	932.48
2M	52.62	1043.60	1096.22
3M	484.72	1225.54	1710.26
4M	1295.91	1691.51	2987.42
5M	1638.83	1794.86	3433.69

Streamline visualization of this data set is shown in Figure 8. Data sets are stored on disk in binary format. For each data set, one hundred streamlines are constructed by using the three programs. The maximum number of time steps for each streamline is set to 5,000.

An IBM RS6000 Model 560 workstation was used for the tests. This machine has 128 megabytes of main memory and 512 megabytes of paging space. Two costs are measured by using wall clock time in seconds. The first one is the initialization cost which is mainly the time to read in the test data. The second one is the cost of constructing 100 streamlines. The total cost is then calculated by adding these two. The tests results are summarized in Figure 9 in which logarithmic scale is used for the y-axis so that very large and small numbers can be plotted in the window. The time breakdown of each case is listed in Table 1, 2 and 3.

Compared with the two virtual-memory-based programs, the performance of the out-

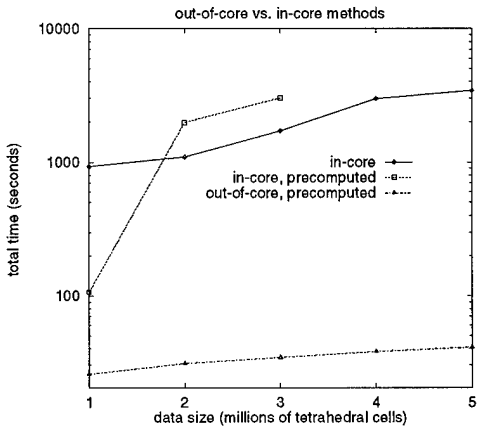

Figure 9: Out-of-core versus vm-based methods.

Table 2: VM-Based Method 2

data size	initiate	construct	total
1M	25.90	80.98	106.88
2M	1367.86	608.55	1976.41
3M	2163.50	866.01	3029.51
4M	–	–	–
5M	–	–	–

Table 3: Out-Of-Core Method

data size	initiate	construct	total
1M	1.71	23.77	25.48
2M	6.60	24.15	30.75
3M	10.14	24.16	34.30
4M	13.62	24.37	37.99
5M	16.44	24.46	40.90

of-core program is up to almost two orders of magnitude better. Its initialization cost grows more slowly with the data size. Its streamline construction cost is small and about constant. The virtual-memory-based programs try to keep as much as data in the main memory during streamline construction. Our test results show a lot of time devoted to allocating memory space and reading in data sets. When the data size is equal to two million cells, the initialization cost grows dramatically, since the size of required memory space already exceeds the size of the physical main memory space. The operating system has to swap out data to the paging space to create memory space for the input data. This situation becomes worse when the data size is increased to three million cells. The second program can not handle the data set with four million cells. The operating system signals a system error and quits the program before the initialization stage is completed. Therefore, a $-$ symbol is shown in Table 2.

The first virtual-memory-based program requires less memory space so it can handle larger data sets. However, the coefficients of the coordinate transformation functions are computed on the fly during streamline construction. Consequently the cost of constructing streamlines for this program is very high compared with the other two programs. The initialization costs of this program are tolerable when the data size is under three million cells. Once the data size reaches four million cells, the initialization cost becomes too high. The total cost is equal to 49 minutes and 47 seconds in this case. For the data set with five millions of cells, this program needs totally 57 minutes and 14 seconds to construct 100 streamlines, while the out-of-core program consumes less than 41 seconds to perform the same operation. Therefore, the performance of the first and the second programs is not acceptable for interactive visualization.

From the above test results, two important findings are: First, the virtual memory system of the operating system is not very helpful for this visualization application. Second, the speed of constructing streamlines is severely degraded if the coefficients of the coordinate transformation functions are not pre-computed. To achieve interactive visualization, there is no doubt that we must trade space for time; in this case, we use less expensive disk space and employ a memory management policy tailored to the streamline calculations.

5.2 The Maximum Octant Size, Size of the Memory Space, and The Block Size Level

Three parameters influence the performance of the out-of-core program. They are the maximum octant size, the size of the memory space, and the block size level. Tests are conducted on the Sun workstation to explore how these three parameters affect the performance of the out-of-core program and to find an optimal combination of the three parameters. In the tests, the maximum octant size is set to 10,000, 20,000, 30,000, and 40,000 cells respectively,

where each cell is represented by 80 bytes of information as explained in Section 5.1. The sizes of memory space are set to 4, 6, 8, and 10 times the maximum octant size. The block size level varies from 1 to 8. The tests are performed as follows:

For each value of the maximum octant size do:

- *Subdivide the data set based on the maximum octant size.*
- *For each memory space size do:*
  - *Create memory space.*
  - *For each block size level do:*
    1. *create free space pool based on the block size level.*
    2. *construct 100 streamlines.*
    3. *measure and report the cost.*

For convenience, a smaller data set of 1.78 million cells is used in these tests. This data set comes from a wind tunnel simulation. Visualization results are shown in Figure 10. Note that the streamtubes are software rendered. The computational cost for the data partitioning and the preprocessing together is about 20 minutes on the same workstation. Note that this cost depends on the maximum octant size. The data is stored in a local disk of the workstation. The initial points of the 100 streamlines are randomly selected. The maximum number of time steps of a streamline is 5,000.

The test results are shown in Figures 11, 12, 13 and 14. The costs of constructing streamlines by using the same maximum octant size are shown in each individual figure. The curves plotted in each figure represent the costs of using different sizes of main memory space while varying the block size level.

By comparing the test results, we can conclude that (for this dataset) the maximum octant size is the most essential parameter in the out-of-core program. The performance of the program is significantly improved when this parameter is reduced from 30,000 cells to 20,000 cells. The out-of-core program favors smaller octant size. It is obvious that the costs decline when the memory space is increased from 4 times to 6 times of the maximum octant size, no matter what the maximum octant size is. However, *further increasing the memory space does not improve the performance.* If the memory space is just 4 or 6 times of the maximum octant size, the out-of-core program performs better when the block size level increases. No significant improvement can be obtained by changing the block size level if the memory space is larger. With our current setting, the best performance is thus obtained when the maximum octant size is set to 10,000 cells and the memory space used is 6.4 megabytes, which is equivalent to 6 times the maximum octant size, and the block size level is 3. The cost of constructing 100 streamlines is below 25 seconds.

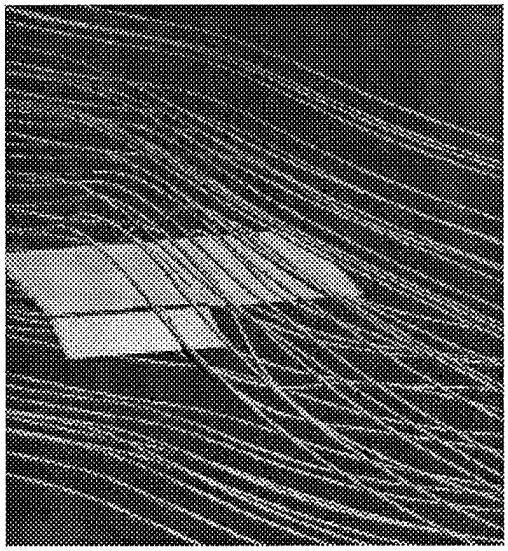

Figure 10: Streamline visualization of the wind-tunnel data set.

In summary, *the out-of-core program performs better when the maximum octant size is smaller, the allocated memory space is larger, and the block size level is higher.* Nevertheless, the improvement made by changing these three parameters has its limits. The reasons can be described as follows. In streamline construction, only a small portion of cells are visited by the streamlines in an octant or even in the whole data set. The performance can not be improved by just loading a larger number of cells into the main memory. Instead, it is improved by loading those cells which are actually used in the integration of a streamline. By using smaller maximum octant size, higher block size level and larger memory space, more octants can stay in the main memory and the percentage of cells which are directly involved in the integration becomes higher. Then more computation can be accomplished between two consecutive octant-fetchings. The overhead of fetching-data is reduced. However, if too many octants are read, the overhead of octant-fetching becomes high. The increase in overhead then cancels out the gain from more local computations, and the performance will reach its limit.

5.3 Average Cost and Overhead

Another set of tests are conducted on the Sun workstation to measure the overhead caused by data-fetching and streamline scheduling and to study how the overhead affects the behavior of the program. A data set of 4.8 millions of tetrahedral cells is used. This data set is

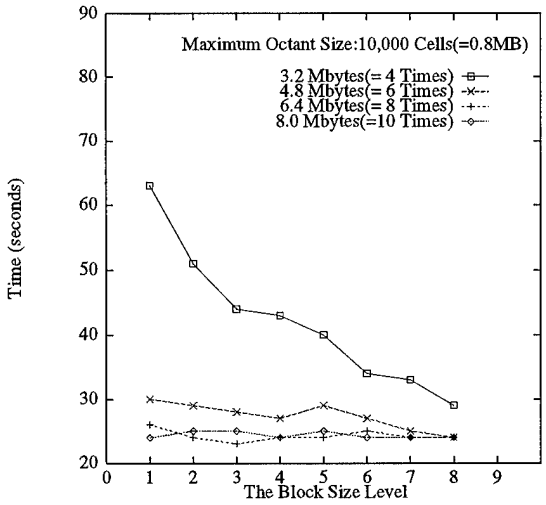

Figure 11: Timing of Program, Maximum Octant Size=10,000 Cells

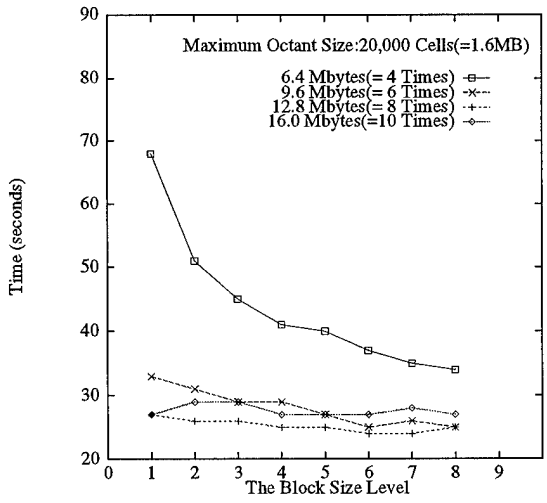

Figure 12: Timing of Program, Maximum Octant Size=20,000 Cells

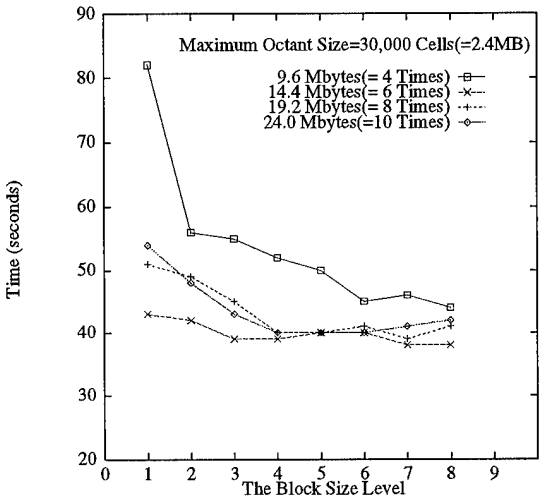

Figure 13: Timing of Program, Maximum Octant Size=30,000 Cells

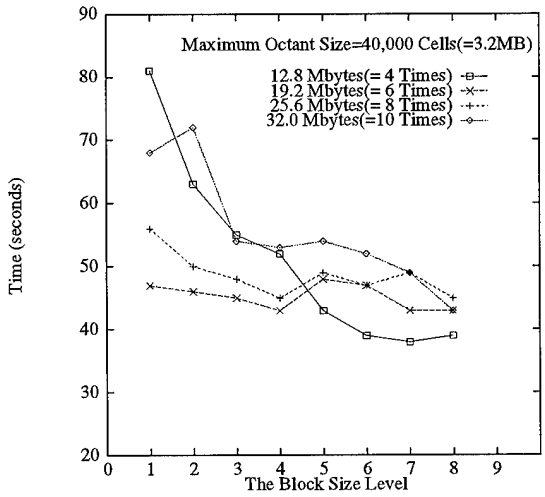

Figure 14: Timing of Program, Maximum Octant Size=40,000 Cells

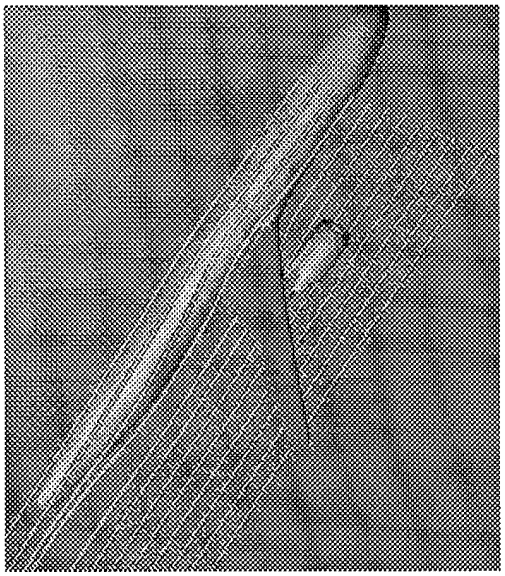

Figure 15: Streamline visualization of the airplane data set.

obtained from a computational fluid dynamics simulation for the flow passing around an airplane body. Visualization results are shown in Figure 15. Note that only a portion of the airplane is modeled. About 407 megabytes of memory are required to store all the vertex and the cell records of this data set. The maximum octant size, the memory space size and the block size level are fixed in the tests. The maximum octant size is set to 40,000 cells. The memory space size is four times of the maximum octant size, and the block size level is three.

Test are performed for calculating 10-100 streamlines. Again, the maximum number of time steps of a streamline is limited to 5,000. Both the total cost and the overhead are measured in each test. The total cost includes the overhead and the cost of integrating the streamlines. Test results are presented in Figure 16. The overhead includes the costs of searching and fetching octants, selecting memory blocks, and scheduling streamline objects. The total cost and overhead are divided by the total number of time steps used in the streamline construction to obtain the average cost and the average overhead for a single step computation. The average costs for a single step computation are shown in Figure 17.

Note that the average cost fluctuates in the test cases. This is because the seed points are randomly selected and therefore the length of each streamline varies. Also note that the average cost does not decrease much when more streamlines are constructed concurrently. The increasing overhead due to streamline scheduling and octant searching cancels out most

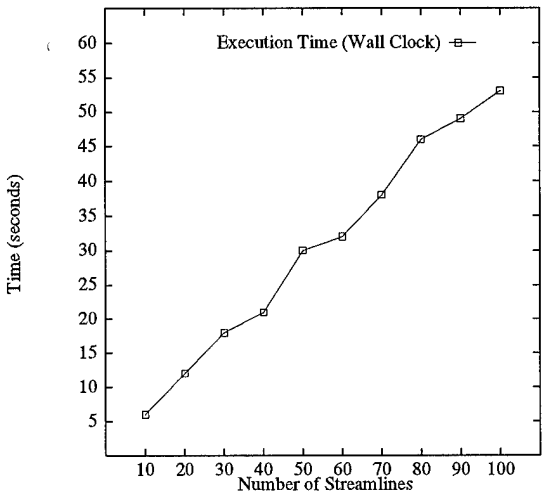

Figure 16: Total Cost of Constructing Streamlines

of the benefit from octant sharing.

Finally, the average overhead is divided by the average cost to produce the percentage of cost due to the overhead. The percentages of cost due to overhead for a single step calculation are depicted in Figure 18. According to the test results, the overhead can be as high as 40 percent of the overall cost.

We also measure the difference in cost of constructing one streamline at a time and multiple streamlines. In one test, one hundred streamlines are constructed one by one. Thus, no streamline scheduling or octant searching is required, and memory allocation is trivial since only one octant and one streamline are resident in the main memory at any time. The average cost of tracing a streamline is about 0.76 seconds under these circumstances. On the other hand, the other test reveals the average cost of constructing 100 streamlines concurrently is about 0.56 seconds per streamline. We observe a 26.3% improvement in performance due to octant sharing, even though overhead is introduced in the multi-streamline execution.

5.4 Local Disk versus Non-Local Disk

In our previous tests, all data is stored in a local disk of the workstations. In some environments, the data may be stored in a non-local disk of a file-server, which is connected with the workstations via a network. In order to explore the effect of storing data in the non-local

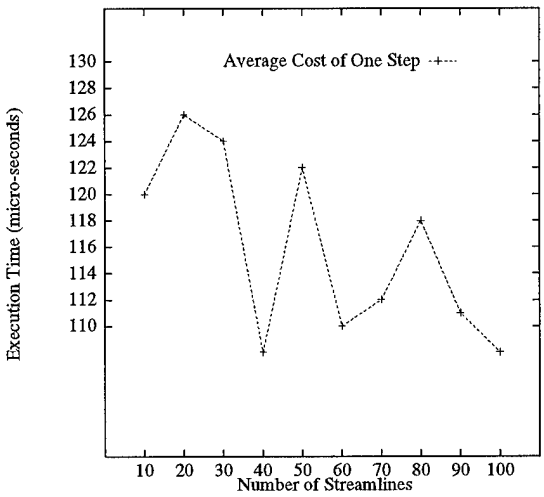

Figure 17: Average Cost of a Single Step Computation

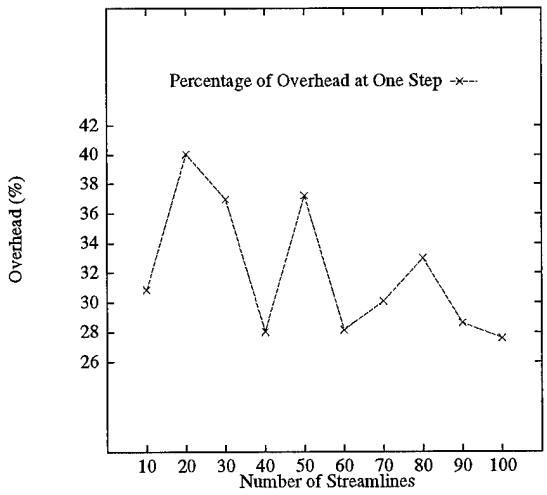

Figure 18: Average Overhead of a Single Step Computation

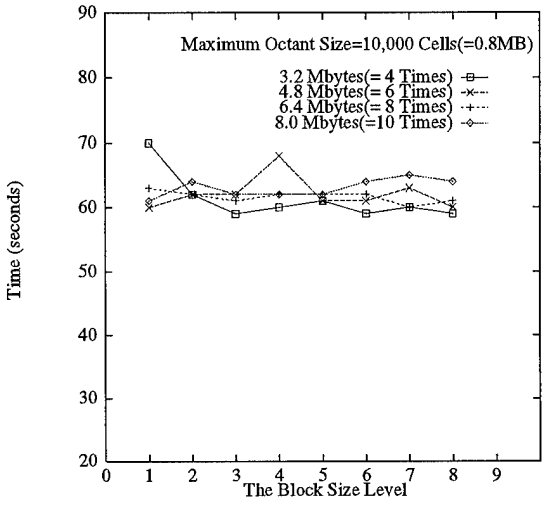

Figure 19: Timing of Constructing Streamlines by Using Non-Local Disk

disk, we set up another set of tests. We repeat the tests described in Section 5.2 by using the same data set. However the data is stored in a non-local disk. Two sets of test results are presented in Figures 19 and 20 and the penalty of using a non-local disk is apparent.

The latency of the network significantly affects the overall performance. The percentage of the cost resulting from the network latency is about 59 to 65% when the maximum octant size is 10,000 cells. It increases to 68 to 76% when the maximum octant size is 40,000 cells. The total cost is increased by at least 100%. Since the network is shared by several computers, the program performance is very unstable. In general cases, the program performs better when the maximum octant size is smaller. This is similar to the results reported in previous tests. Again, the performance does not improve when more memory space is allocated. This is because octant fetching becomes more frequent in order to fill the additional memory space, which triggers more non-local disk I/O.

6 Conclusions

We have presented an efficient out-of-core algorithm for visualizing very large unstructured vector field data sets on a single workstation with only moderate size of main memory. Using an octree structure, the data sets are partitioned into subsets and stored in disk files. These subsets are read into the main memory on demand and a memory management policy is

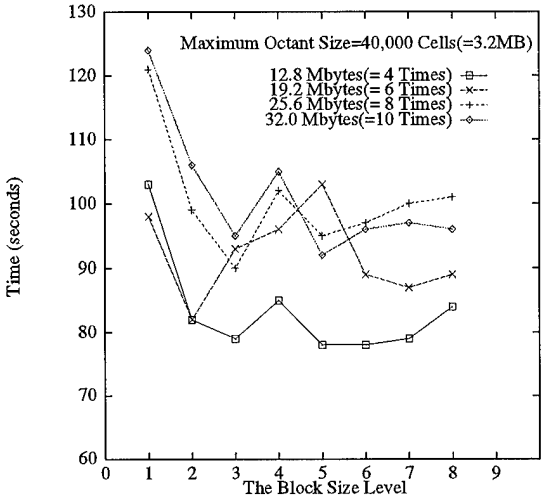

Figure 20: Timing of Constructing Streamlines by Using Non-Local Disk

designed to allocate memory space for storing them. Tests are conducted to explore the performance of the algorithm and its implementation.

Test results demonstrate that the out-of-core algorithm enables interactive streamline visualization of data sets of several millions of tetrahedral cells on an average workstation. For example, for a data set with 1.78 million cells, the computational cost for constructing 100 streamlines concurrently, each of them with as many as 5,000 integration points, is below 25 seconds on a Sun SPARC-20 while only using 6.4 megabytes of its main memory space. We also show that the same visualization requirements cannot be achieved using virtual memory.

The use of a high-end workstation like a Sun Ultra SPARC or an SGI Indigo2 would further increase the interactivity. The test results reveal that the performance of our program is better when the data division is finer, block size level is higher, and the memory space used is larger. We also show that the out-of-core program runs much faster when the data are stored in a local disk.

Future work includes making use of the hardware rendering capability on a graphics workstation, optimizing the preprocessing step and designing out-of-core algorithms for other types of visualization operations, such as surface and volume rendering.

Acknowledgment

This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-19480, and by the National Science Foundation under the ACERC Center. Thanks to Tom Crockett for very constructive suggestions.

References

[1] Final Progress Report for Phase I SBIR: A Software Architecture for Efficient Visualization of Large Unsteady CFD Results, July 1996. FieldView, Intelligent Light.

[2] DARMOFAL, D., AND HAIMES, R. Visualization of 3-D Vector Fields: Variations on a Stream, Jan. 1992. AIAA Paper 92-0074 (AIAA 30th Aerospace Science Meeting and Exhibit, Reno, NV).

[3] HAIMES, R. pV3: A Distributed System for Large-Scale Unsteady CFD Visualization, Jan. 1994. AIAA Paper 94-0321 (AIAA 32th Aerospace Science Meeting and Exhibit, Reno, NV).

[4] HAIMES, R., AND BARTH, T. Application of the pV3 Co-Processing Visualization Environment to 3-D Unstructured Mesh Calculations on the IBM SP2 Parallel Computer, Mar. 1995. presented at CAS Workshop, NASA Ames Research Center.

[5] HULTQUIST, J. P. M. Constructing stream surface in steady 3d vector fields. In *Proceeding of Visualization '92* (1992), IEEE Computer Society Press, pp. 171 178.

[6] KENWRIGHT, D. N., AND LANE, D. A. Interactive Time Dependent Particle Tracing Using Tetrahedral Decomposition. *IEEE Transactions on Visualization and Computer Graphics 2*, 2 (1996), 120 129.

[7] KENWRIGHT, D. N., AND MALLINSON, G. D. A 3-d streamline tracking algorithm using dual stream functions. In *Proceeding of Visualization '92* (October 1992), IEEE Computer Society Press, pp. 62 68.

[8] KINCAID, D., AND CHENEY, W. *Numerical Analysis.* Brooks/Cole Publishing Company, 1991.

[9] LOHNER, R. A Vectorized Particle Tracer for Unstructured Grids. *Journal of Computational Physics 91* (1990), 22 31.

[10] SAMET, H. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, Inc., 1990.

[11] SCHROEDER, W. Streaming Pipeline, October 1996. personal communication.

[12] SILBERSCHATZ, A., AND GALVIN, P. B. *Operating System Concepts.* Addison-Wesley Publishing Company, 1994.

[13] UENG, S.-K., SIKORSKI, K., AND MA, K.-L. Fast Algorithms for Visualizing Fluid Motion in Steady Flow on Unstructured Grids. In *Proceedings of Visualization '95* (1995), IEEE Computer Society Press, pp. 313 319.

[14] UENG, S.-K., SIKORSKI, K., AND MA, K.-L. Efficient Streamline, Streamribbon, and Streamtube Constructions on Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics 2*, 2 (1996), 100 110.

[15] VENNARD, J., AND STREET, R. *Elementary Fluid Mechanics.* John Wiley & Sons, Inc., 1975.